advanced research computing

Effective Cluster Use

(by non-programmers)

Andy Gittings

ARC - Scientific Research Software Advisor

support@arc.ox.ac.uk

Please ensure your Video and Audio Streaming are switched OFF

Use chat window for questions

IT services

Why High Performance Computing?

High Performance Computing

Scientific computing is the "third pillar" of the empirical sciences

- provides quantitative solutions to mathematical models
- typically requires vast quantities of floating point operations
- **for a given algorithm**, computing performance (how fast simulations can run) determines the scale and complexity of the scientific numerical problems we can solve

How fast is fast enough?

- a PC can deliver tens of Gflops (flops = **fl**oating point **o**perations **p**er **s**econd)
- tens of millions is a lot of flops but that may or may not be enough
- an extreme example: **short range weather forecast**
  - prediction for next day(s) must be delivered in less than 1 day
  - the Met Office models translate into a requirement for ~1 Pflops
  - that is 1 million times more that the PC, so the simulation has to run on many CPUs, working together on the same model
- the fastest **supercomputer** (November 2019) achieves almost 200 Pflops (and consumes 10MW power)

High Performance Computing

High Performance Computing (**HPC**)

- the practice of programming and running **scientific applications** efficiently on **supercomputers** (as well as **modern computers**)
- is equivalent with **parallel computing**.

Supercomputers are

- computer systems at the frontline of processing capacity (at any one time)
- designed mainly for **parallel computing**

Traditional role of **parallel computing** (from early 1990s):

- **more CPUs is better**: the ability to tackle a large problems by using many processors working concurrently on parts of the problem in a coordinated fashion
- the CPU **work load** and **the memory** usage are spread across parallel resources

Recent role of parallel computing (from mid 2000s):

- new CPUs are faster only because they have multiple cores, so applications must make use of them through data/task parallelism to run any faster

Multicore CPU design

Moore's Law (1965):

- the number of transistors in CPU design doubles roughly every 2 years
- backed by clock speed increase and instruction-level parallelism (ILP), this has correlated with exponentially increasing CPU performance for at least 40 years

This meant the same old (single-threaded) code just runs faster on newer hardware. No more! While the "law" still holds, clock frequency of general purpose CPUs was "frozen" in 2004 at around 2.5-3.0 GHz and design has gone multicore.

Why? Because CPU design has reached the **walls** (D. Patterson, Berkeley report)

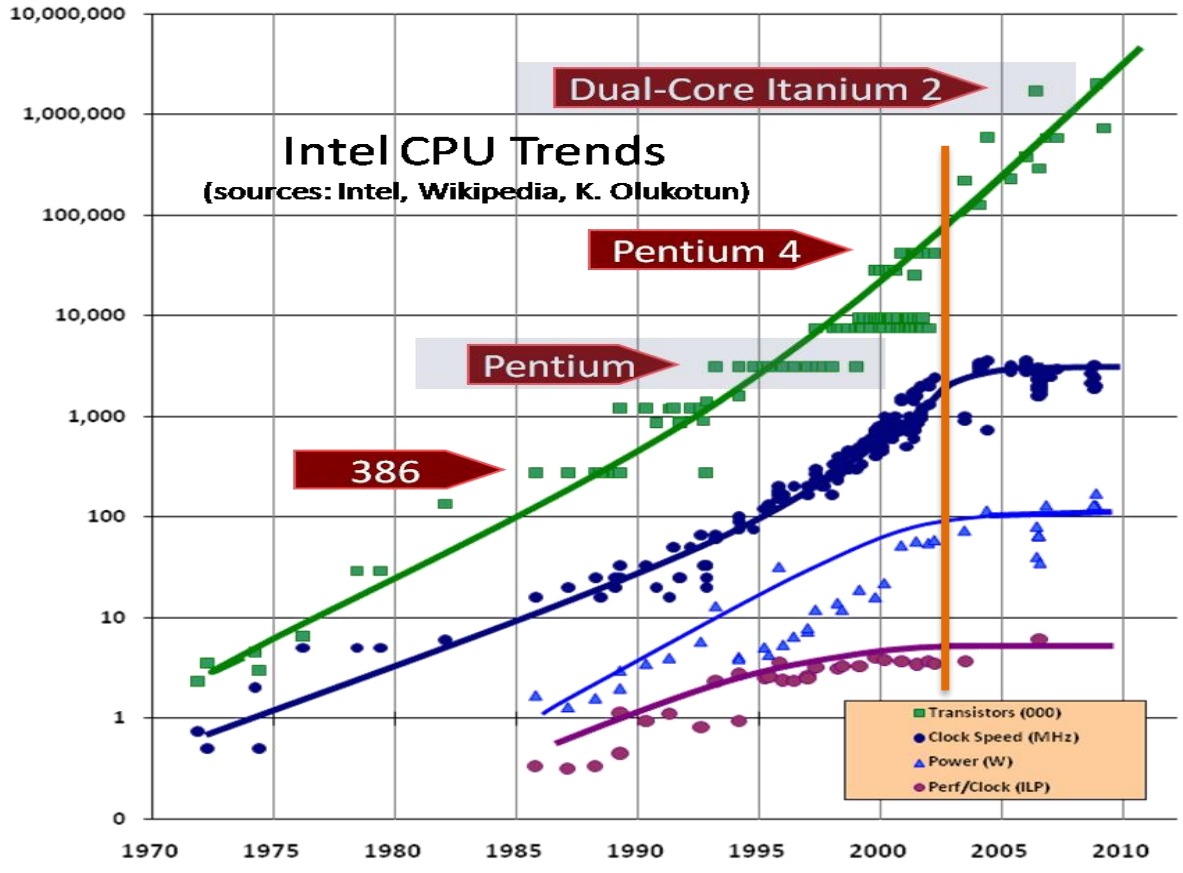
- **power wall** (power increases with clock speed, faster and smaller CPUs get too hot)
- **memory wall** (increasing gap between CPU and memory speed, masked by caching)
- **Instruction-Level Parallelism (ILP) wall** (a deeper instruction pipeline is of limited use and means more power)

Multicore design addresses the power wall and posed new problems to programmers.

Multicore CPU design

Drastic turn in May 2004:

- Intel ditched the Tejas CPU design, projected to run at 7GHz and draw 150W
- changed the direction towards multicore design



Multicore CPU design

The industry “promise” of increased speed (through thread-level parallelism):

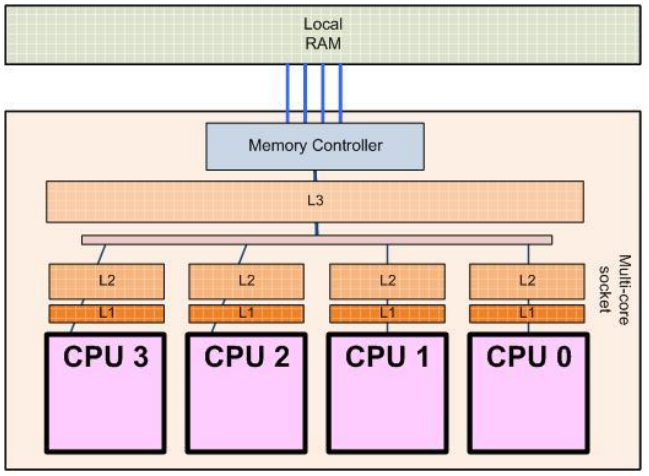
- multiple cores replicated on the same die, overall performance ~ #cores X core performance

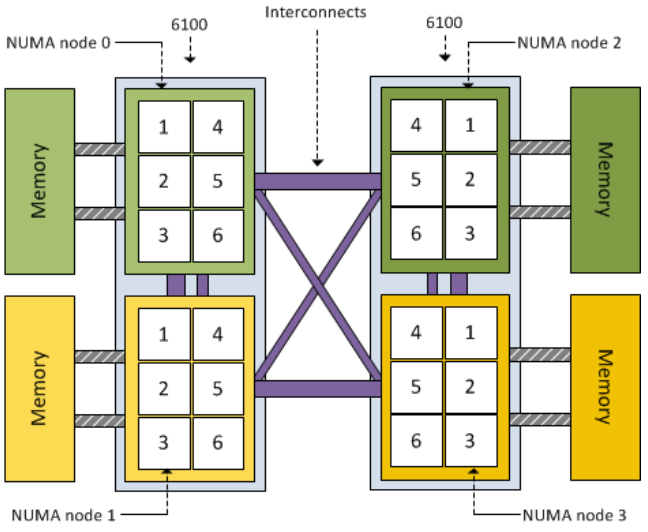
Possible, but design limitations (inc. cost) dictate that

- shared resources: cache levels, memory controller, IO hub
- memory access is non-uniform (NUMA)
- memory and IO bandwidth are limiting factors

Main challenge for programming is communication, which can directly determine performance:

- computation is cheap, memory access/IO costly
- compute bound problems become memory intensive and big data can turn a compute bound problem into an I/O bound one
- “starved” CPUs means low performance





Parallel computing is for all

For the past decade, the trend in supercomputer design has been towards

- **clusters** (compute servers connected by a fast local network) and
- **commoditisation** (off-the-shelf components, economies of scale)
  - x86 is the dominant instruction set (Intel and AMD CPUs)
  - Linux is the dominant OS

At the same time, CPU design incorporates more and more cores.

The difference between your departmental machine and a supercomputer is limited to

- server grade hardware (not much faster but more reliable, maybe more RAM)
- fast (usually Infiniband) network and fast parallel storage

HPC (traditionally the preserve of high-end computing in physics and engineering) has become a **challenge for everyone** (software vendors, researchers, etc.) **everywhere**

- **supercomputers** as well as
- **servers** and **workstations** (with multicore CPUs).

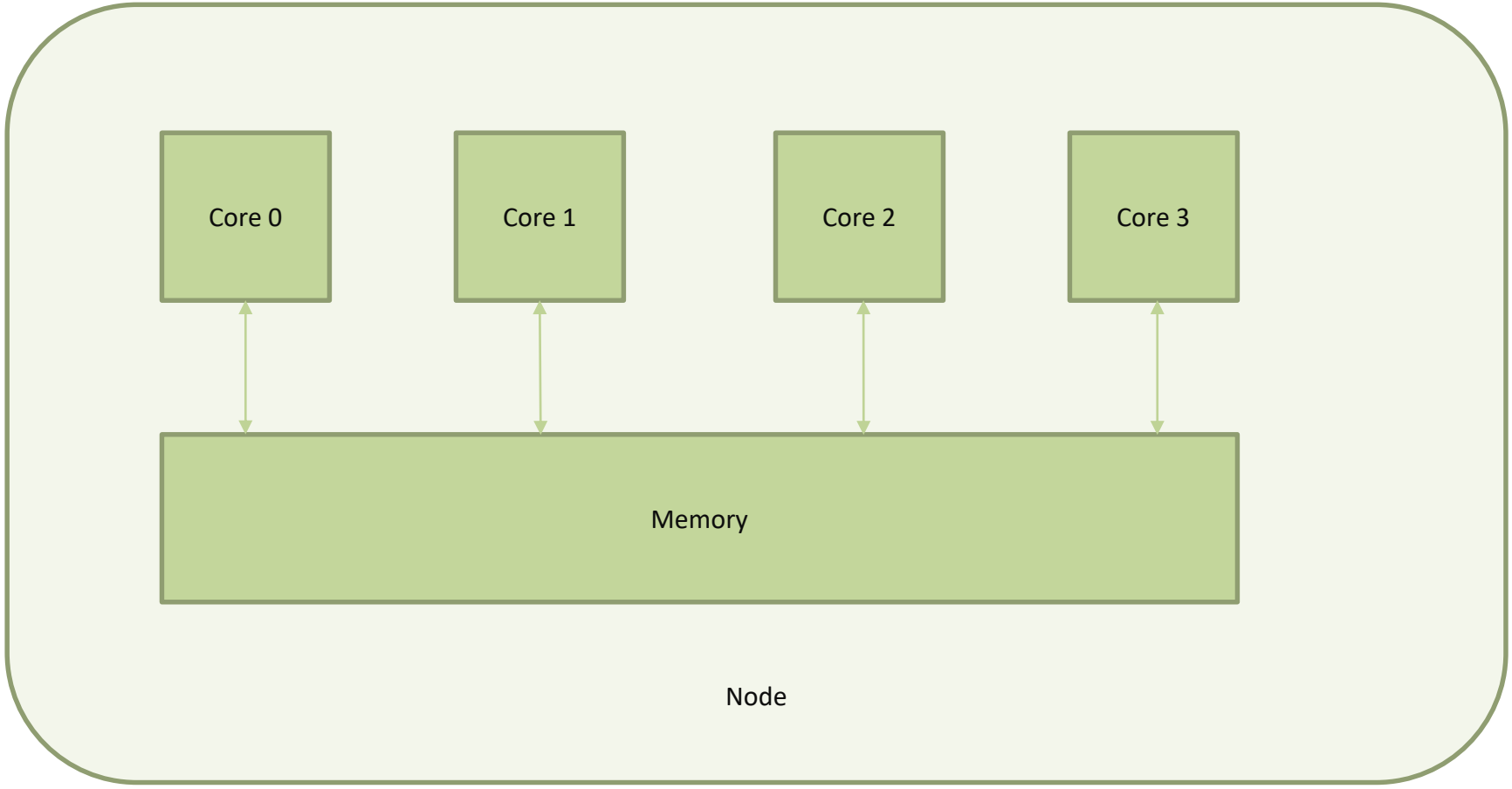
Before we start…

- Glossary
  - **core** = unit that does the work (sometimes use CPU as a synonym)
  - **processor** = collection of cores in a single package all sharing the same memory
  - **node** = a collection of processors all sharing the same memory
  - **interconnect** = the network in a machine that joins together the separate nodes

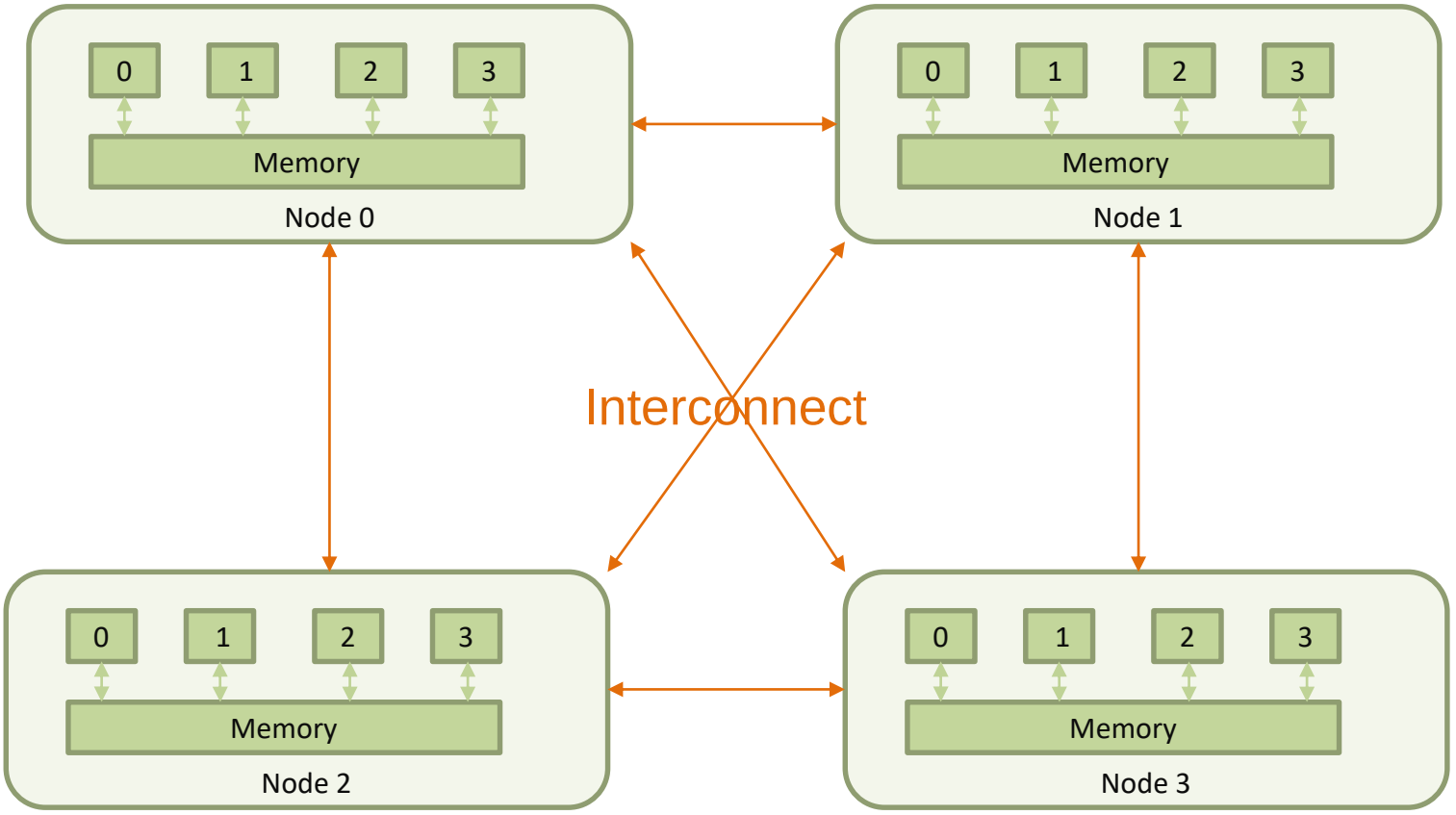
Note: each node has its own memory and cannot directly "see" another node's memory.

- Distinction between processor, process and thread
  - **processor** = a physical piece of **hardware**
  - **process** = an instance of a running program (**software**)
    - essentially it has two components: instructions to execute and associated data
    - in parallel programming we often have multiple instances (processes) of the same program…
  - a process always consists of one or more **threads** of execution

Could be your laptop



Or a simple cluster



Login Node
(Editing, Compilation, Job Submission etc.)

Introduction to Parallel Computing

Who needs parallel computing?

Researcher 1	• has a large number of independent jobs (e.g. processing video files, genome sequencing, parametric studies) • uses serial applications • needs to accelerate research	**High Throughput Computing (HTC) (many computers):** • dynamic environment • multiple independent small-to-medium jobs • large amounts of processing over long time • loosely connected resources	
Researcher 2	• developed serial code and validated it on small problems • to publish, needs some "big problem" results • hits a performance wall		**High Performance Computing (HPC) (single parallel computer)** • static environments • single large scale problems • tightly coupled parallelism
Researcher 3	• needs to run large parallel simulations fast (e.g. structural mechanics, molecular dynamics, computational fluid dynamics)		

How can the ARC help you?

Researcher 1	• has a large volume of independent jobs (e.g. processing video files, genome sequencing, parametric studies) • uses serial applications • needs to accelerate research	**HTC**: • large number of jobs • one job runs on one cluster node • one job can harness the 16 cores available per node – this is achieved through multi-threading or via coordinating concurrent processes
Researcher 2	• developed serial code and validated it on small problems • to publish, needs some "big problem" results • hits a performance wall	**Parallel code development**: • parallel execution may be a solution • this course can be a start • seek ARC advice
Researcher 3	• needs to run large parallel simulations fast (e.g. structural mechanics, molecular dynamics, computational fluid dynamics)	**HPC**: • single large scale problems • jobs run on multiple nodes • many commercial and free applications can be run across multiple nodes

Parallel Processing Models

Models of parallelism:
Distributed Memory

Distributed Memory Programming Model:

- multi-core system, each core has its own private memory
- local core memory is invisible to all other processors
- agent of parallelism: the **process** (program = collection of processes)
- exchanging information between processes requires **explicit** message passing
- the dominant programming standard: **MPI**

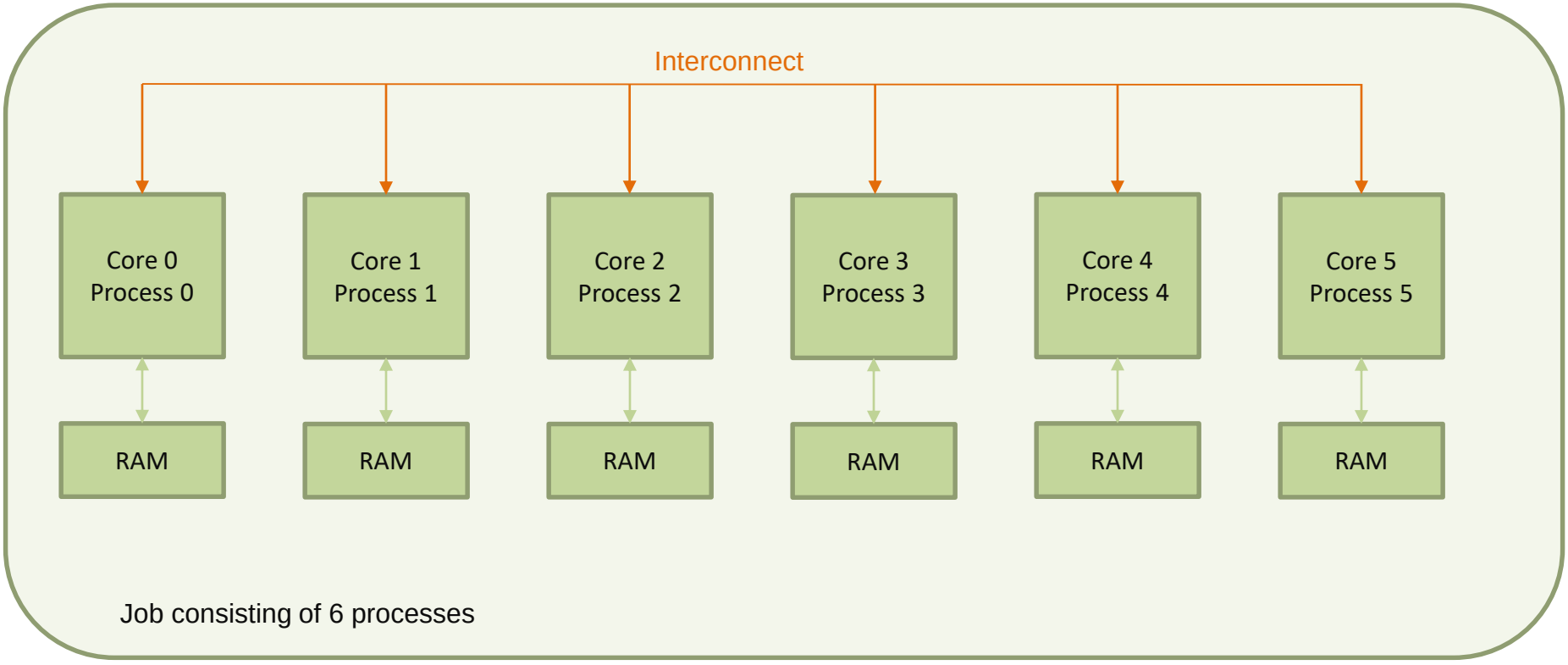
Distributed Memory Hardware:

- conceptually, many PCs connected together (traditional Beowulf cluster)
- current approach:
  - multi-core computer nodes (high-density blades) with own memory
  - high-bandwidth, low-latency network connection
  - off-the shelf modular technology (high-end CPUs, standard hard disk)
  - accounts for the largest HPC systems

Distributed Memory ARC systems: the **arcus-b** cluster (but any machine can be programmed using this model)

And in fact we will use this during one of the practical exercises to run DL_POLY

Distributed Memory View



Models of parallelism: **Shared Memory**

Shared Memory Programming Model:

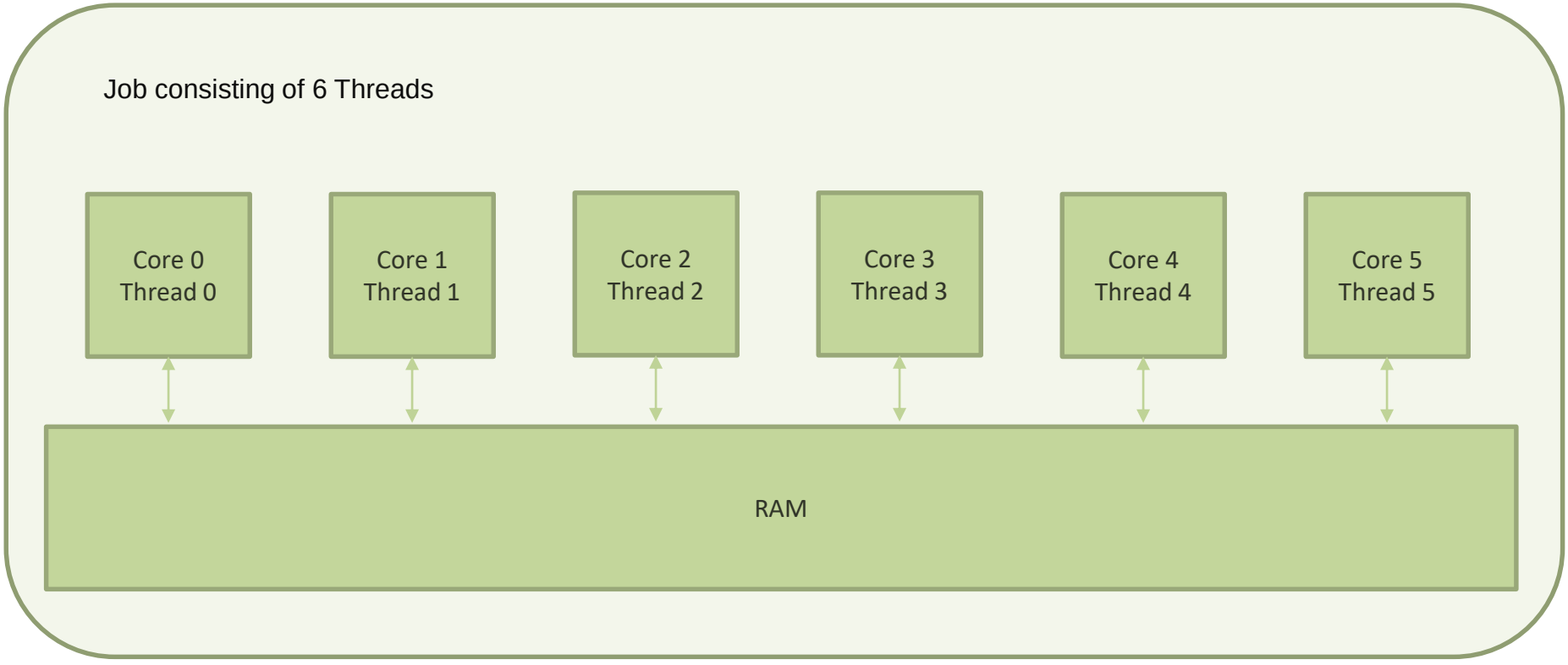
- multi-core system
- each core has access to a shared memory space
- agent of parallelism: the **thread** (program = collection of threads)
- threads exchange information **implicitly** by reading/writing shared variables
- the dominant programming standard: **OpenMP**

Shared Memory Hardware:

- conceptually, a single PC, with a large memory and many cores
- accounts for both small and inexpensive systems (desktops) and very large and expensive system (with very expensive high bandwidth memory access)

Shared Memory ARC Systems: **arcus-HTC** and any *single node* of the **arcus-b** cluster.

Shared Memory View



Distributed Memory v. Shared Memory

- Distributed Memory:
  - **Can scale to any number of cores**
  - Requires special tools to compile and run the code
    - Typically `mpicc` or `mpif90` to compile, `mpirun` to run it
  - Can be harder to program that shared memory
  - But will generally perform better if done well
  - And it teaches good parallel programming “habits”
- Shared memory
  - **Is usually limited to the number of cores in a node**
    - Can overpopulate, good for debug, bad idea for performance
  - Generally just requires an extra flag on the compiler
  - Can be easier to program than distributed memory
  - It is often hard to get good parallel performance
    - Sharing things is not good for parallelism …
  - Can easily let people be a bit sloppy when programming …

Batch Scripts

Batch Scripts

- Now we know about the structure of a cluster and the nature of jobs that you can run on it we can think about how to structure our batch script to best take advantage of it
- For the clusters provided by ARC we need to know
  - Each node has a *minimum* of 16 cores
  - In **arcus-b** you are charged PER NODE – this is the most common form of charging
    - So if you only use a subset of the cores you will still get charged for all of them (capped at 16 cores)
    - You get the node in exclusive node
    - You always can use all the memory in the node
- Remember to adjust the following as appropriate if you use machines elsewhere!
- Also remember the first part of the script reserves resources for you, while the second says what you want to do with it

Some quick solutions - HTC

Example for HTC-type job script (researcher 1):

- serial (or multi-threaded) application
- parametric study, many input files
- processing in batches of 16
- each job uses 1 compute node
- ideally, processing should be balanced

```
#!/bin/bash
…
#SBATCH –walltime=10:00:00
#SBATCH --nodes=1
…
for ID in {1..16}; do
  serialApp test_$ID.dat &
done
…
wait
```

Reserve 1 node for 10 hours

Run 16 jobs in the background

Wait for all the jobs to complete

Some quick solutions - HPC

Example for HPC-type job script (researcher 3):

- parallel (MPI) application
- single large problem, too large for single node
- one single input file
- job uses many compute nodes
- For best resource usage use multiple of 16 cores

```
#!/bin/bash
…
#SBATCH --nodes=2
#SBATCH --ntasks-per-node=16
…
. enable_arcus-b_mpi.sh
mpirun $MPI_HOSTS parApp test.dat
```

Array Jobs

- Job arrays allow you to submit the same batch script many times over

```
sbatch –array=1-10 myscript
```

- By default you can distinguish between members of the array with the `$SLURM_ARRAY_TASK_ID` environment variable, e.g.

```
#!/bin/bash
…
#SBATCH --walltime=10:00:00
#SBATCH --nodes=1
…
cd job_name.$SLURM_ARRAY_TASK_ID
for ID in {1..16}; do
  serialApp test_$ID.dat &
done
…
wait
```

Array Jobs

- The main use is to allow the HTC user to use more than 1 node
  - However there is no reason why an MPI user can't use them
- And also note there is *no* performance difference from submitting each of the job members individually
- The main reason is convenience
  - Can submit all with one command
  - Can use `scancel` to cancel all the jobs with one command
- Strong recommendation: Don't use very large job arrays, if things go wrong things can go VERY wrong!
  - Do you want emails from each of 10,000 failing jobs all at the same time?

Load Balancing and Array Jobs

- When we pack jobs up into groups of 16 the time taken is determined by the one that takes the longest
- This can cause efficiency problems if one or two of the jobs takes very much longer than the others as you will have to wait for the longest to complete irrespective of how quick the others are
- In other words you want the group of 16 to be *load balanced*
- Not much you can do if you just have 16 jobs
- But if you are using a job array try to make each member of the array as balanced as possible
  - You will generally have some kind of feeling which runs are quick to complete and which are slow, so group the quick with the quick and the slow with the slow
- So a bit of thought can help your efficiency quite a lot!

Short Job? – The Devel Partition

- If your job is 10 minutes or less and 64 cores or less you can use the devel partition for quick turn around

```
sbatch --partition=devel …
```

- Very useful for checking batch scripts work …

- More generally you should set as short a time as you think will be enough for the job to run
  - Short jobs normally have higher priorities

Large Memory Jobs

- Sometimes jobs require lots of memory.
- Possible solutions are
  - Node under population (Generic)
    - Instead of 16 jobs on a node each with an average of 4Gbytes available, run 4 jobs with 16 Gbytes available – obvious cost implications
  - Use large memory nodes if available – ARC has quite a few 128 Gbyte nodes and a small number of 256 Gbyte nodes
    - There is also a 1.5 Tbyte node but this is available by request only

…

#SBATCH –mem=120000

…

Under Populated MPI Jobs

- For MPI jobs that use under population you have to be careful where the processes end up
  - You don't want all the processes on the same node

```
#!/bin/bash

…
#SBATCH --nodes=2
#SBATCH --ntasks-per-node=8

…
. enable_arcus-b_mpi.sh
mpirun -np 16 parApp test.dat
```

- If you use a hybrid MPI/OpenMP application the same considerations apply

Compilers

What else can we improve

- There are a number of other ways to generate your results faster on a cluster:
  - **Make the program run faster by either better use of the compiler or libraries**
  - **Better use of the disks**
  - **Using an appropriate number of cores for your MPI or OpenMP program**
  - *Use of area specific or application specific knowledge*
- The first three we'll discuss now
  - We'll need to know how to measure parallel performance for the second
- The last is a huge area and mostly beyond what we cover today
  - More advanced and application specific courses may address this
  - Note especially for MPI programs there are often application specific "tricks" that can help you obtain your answer more efficiently on a cluster

Programming languages

Programming languages for scientific computing:

- **Fortran** and **C** account for most computation intensive codes
  - computation engines of many applications
  - Fortran is more “natural” than C for scientific computing
    - use Fortran 95 or later, Fortran 77 is dead!
  - performance libraries are written in C (FFTW) or Fortran (LAPACK)
- **C++**
  - OOP allows (it is claimed) better software design and re-use of code
- **JAVA**, **C#**, etc.
  - normally used only for front-ends and GUIs etc.
- **Matlab**, **Python**
  - interpreters, interactive use (data inspection, plotting capabilities)
  - numerically intensive parts written in C/Fortran (mex functions, modules)

Compiled languages

Fortran, **C** and **C++** are *compiled*

- the computer cannot understand the program (human readable) directly
- the *compiler* is a tool used to translate the *whole* program into the instructions that a computer can understand
- there are many ways to do this translation; how fast the resulting program runs will depend upon how "good" a job the compiler does

Compare with **Matlab**, **Python** and **R**

- interpreted languages
- again, a tool is required to turn the program into something the computer can understand, but this is done one "line" at a time
  - easy on the tool and convenient in some ways (*e.g.* what if your program is 1000s of lines long and you change one line only?)
  - but typically *much* lower performance than compiled program

Making most of compilers

So, we want our program to run as fast as possible.

There are two major ways we can affect its performance via the compiler:

- the choice of compiler and
- the use of compiler (*i.e.* the choice of compiler flags)

Compilers

On many systems several compilers are available.

For instance on ARC systems one can use:

- The GNU compiler collection (standard Linux compilers – available everywhere and free) : `gcc/g++/gfortran`
- The Intel compiler suite : `icc/icpc/ifort`
- The Portland Group compilers : `pgcc/pgCC/pgf90`

Choosing the compiler

How to choose which compiler you are using varies from system to system, but a very common method is via *modules*, which are a general method to manage software installations. For instance on ARC systems the following will pick versions of the appropriate compiler:

```
module load intel-compilers
module load gcc
module load pgi
```

(Fortran programmers – note these are entirely separate from and have nothing to do with Fortran modules)

Choosing the compiler

As an example of how the compiler can affect the run time here are two examples of DL_POLY_4 run on 16 cores of Arcus (one single node) with the three different compilers. Versions of the compilers are indicated. Times are in seconds.

Compiler	Sodium Chloride	Gramicidin
gcc 4.8.2	241.686	189.514
Intel 14.0.2	342.201	246.520
Portland Group 13.10-0	**205.602**	**129.827**

Invoking the compiler

- However it's not just which compiler you use, it's also how you invoke it -- this usually makes more difference than the choice of compiler
- So let's have a look at how a compiler works in practice
- There are actually a number of stages, but only two are of interest to us
  - Compilation
  - Linking

Compilation

- Remember one program can be contained in many *source files*
- Compilation is the stage that takes an individual file and translates it into instructions the computer can understand.
- These instructions are placed in an *object file* with a .o suffix
- You can compile only (no linking) by use of the `-c` flag:

```
$ ls
file.f90
$ gfortran -c file.f90
$ ls
file.f90  file.o
```

Linking

- Remember that a program can be in many different source files
- Linking takes all compiled object files and links them together into a single *executable*
- Linking is normally managed through the compiler tool itself (which uses the default linux linker **ld** to do the work)

```
$ ls
file1.f90  file2.f90  file3.f90
$ ifort -c file1.f90
$ ifort -c file2.f90
$ ifort -c file3.f90
$ ls
file1.f90  file1.o  file2.f90  file2.o  file3.f90  file3.o
$ ifort file1.o file2.o file3.o -o exe
$ ls
exe  file1.f90  file1.o  file2.f90  file2.o  file3.f90  file3.o
```

Compile and link flags

- We have already used flags to change the (default) behaviour of the compiler and linker
  - –c specifies “compile only” and –o specifies the executable file
- All compilers have many, many flags
- Similarly, linkers have many flags
  - the most important are those telling the linker where to find files (esp. libraries)
- The usual flags at the compile stage include
  - Help with debugging the program
  - Making sure the programmer sticks to international standards
  - Telling the compiler where to find files – e.g. `-I` for include files
  - *Optimisation flags* (these are the most important flags for us)

Optimisation flags

- All compilers have a flag of the form `-ON` where N is an integer, typically in the range 0-3
- Use of this will make the compiler analyse each file it is working on in an attempt to produce a faster executable code
- The larger the number, the harder it will try to do so:
  - `-O0` – don't optimise the code
  - `-O1` – do quick and easy optimisations
  - `-O2` – try hard to get the best performance
  - `-O3` – try really hard to get the best performance!
- Not quite a free lunch
  - Longer compiler times
  - More likely to show up compiler bugs
  - Also more likely to show up software bugs in strange ways …
- But you should really use the highest of these for productions runs!

What difference does it make?

-O0

Compiler	Sodium Chloride	Gramicidin
gcc 4.8.2	241.686	189.514
Intel 14.0.2	342.201	246.520
Portland Group 13.10-0	**205.602**	**129.827**

-O3

Compiler	Sodium Chloride	Gramicidin
gcc 4.8.2	115.956	84.919
Intel 14.0.2	**99.942**	**76.556**
Portland Group 13.10-0	108.292	78.945

Other optimisation flags

- All compilers have many optimisation flags
- Unfortunately apart from `-O` they are almost always specific to the compiler
  - You will have to look at the **man** page or the user guide
- Some suggestions (apply to all languages we have considered):

```
gcc -O3 -funroll-loops -march=native …
icc -O3 -xHost -ipa …
```

- ipa = **i**nter-**p**rocedural **a**nalysis, analyses **all** source code (not just one file at a time), looking for optimisation opportunities **across** files
- `-ipa` can MASSIVELY increase compile time

Optimisation Flags

gcc	NaCl
`-O0`	241.686
`-O1`	132.374
`-O2`	120.537
`-O3`	115.596
`-O3 -funroll-loops`	119.578
`-O3 -funroll-loops -march=native`	106.882

icc	Compile time	NaCl
`-O0`	65	342.201
`-O1`	142	114.795
`-O2`	237	99.447
`-O3`	267	99.942
`-O3 -xHost`	285	96.769
`-O3 -xHost -ipa`	4202	97.516

Demonstration

In Practice – the “Makefile”

You don’t always compile the whole program from the command line

- Often something called a Makefile is supplied which will automate the build process.
- How to set the compiler and compiler flags in the case will vary from case to case

However …

- Commonly you set a variable called CC to the name of the C compiler
- And one called FC or F90 for the Fortran compiler
- The C compile flags are usually called CFLAGS
- And the Fortran compile flags FFLAGS, FCFLAGS or F90FLAGS

Example Makefile

```
# serial compiler
CC      = icc
# compiler flags
CFLAGS  = -O2 -xHost -Wall
# include files
INC     = -I$(MKLROOT)/include
# libraries
LDFLAGS = -L$(MKLROOT)/lib/intel64 -openmp -mkl=parallel -lpthread -lm

# rules
.SUFFIXES:
.SUFFIXES: .c .h .o

.c.o:
                $(CC) $(INC) $(CFLAGS) $(COPTS) -c $<
.DEFAULT:       blas

blas:           blas_demo.o blas_demo_aux.o
                $(CC) $(CFLAGS) $(COPTS) -o blas_demo blas_demo.o
blas_demo_aux.o $(LDFLAGS)
```

Compiling MPI programs

- Some parallel programs use MPI
  - As discussed already
- These should be compiled using the MPI wrapper for the compiler
  - Usually called mpicc/mpif90
- This takes exactly the same flags as the normal invocation of the compiler
  - In fact all it really is is the normal invocation with a few extra flags added for you!

Linker flags

- Similar to the compile stage the linker can also use many flags
- By far the most important of these for us are
  - -o which names the executable
  - flags to tell the linker where to find “extra” object files
- This last point takes us towards libraries, our next point
- Libraries allow us to use very efficient code that somebody has already written
  - And so more efficiently use the cluster

Libraries

- So libraries allow programmers access to very efficient code for commonly performed operations
- For instance MKL (the Math Kernel Library) is Intel's library that performs linear algebra very efficiently on Intel chips
  - Matrix-Matrix Multiplication, Linear Equation Solves, Eigenvalue problems, SVD, etc.
- However their use is again a little bit beyond what we can cover here
  - Though there is an optional exercise to investigate their use
- But there is one thing to remember

Use the Centrally Installed Libraries!

- When we install software on ARC systems we always try to install software using the best performing libraries
  - Sometimes this is quite tricky
- This is why you should always use the modules we provide
  - Unless you have a very specific need
- The performance of python, MATLAB, R … really depends on these – use the centrally installed software if at all possible and don't install your own

Useful Libraries - BLAS and LAPACK

- There are many useful libraries for Scientific computing and I'll mention a few over the next few slides
- Possibly the most important are
  - BLAS – Basic Linear Algebra Subprograms
  - LAPACK – Linear Algebra Package
- Reference versions are available from www.netlib.org
- However you ***should not*** use these
- Rather you should use one of the optimised implementations
  - MKL on Intel
  - ACML on AMD
  - ATLAS – Portable, optimised BLAS. Pain in the butt to install …
  - OpenBLAS – Portable, optimised BLAS, continuation of GotoBLAS
- On ARC machines as they are intel chips MKL is provided
  - `module load intel-mkl`

Other useful Scientific Libraries

- FFTW – the *de facto* method for ffts – www.fftw.org
- Boost – libraries for C++ programmers
- GSL – GNU Scientific Library – www.gnu.org/software/gsl
- ScaLAPACK – distributed memory version of LAPACK – in MKL
- NetCDF and HDF5 – libraries to make input/output easier and data more portable

Parallel libraries

- Some libraries are parallel
  - They can use multiple cores to accelerate the computation
- ScaLAPACK is distributed memory
  - To use it requires code changes
- But many implementations of BLAS and LAPACK can use shared memory parallelism
- So we can use this to make our calculations faster without changing the code
- We will investigate this also in the practical

Libraries - summary

- Libraries provide a method for easy-ish use of highly efficient code produced by many different programs
- They also allow access to functionality that may be difficult to implement (e.g. FFT, eigensolver)
- Many modern libraries can use parallelism, so you can parallelise the code just by linking in an appropriate library
- Use the centrally installed software!

Use of Disk

Use of Disk

- Getting the best out of many applications depends on getting the best out of using the filesystem where the files the applications uses are read from or written to
- How to best use the filesystem is cluster specific, but what is best for ARC often can be adapted with only small changes for other clusters
- There are 2 main issues
  1. Amount of I/O – i.e. using lots of disk
  2. Efficiency of I/O – i.e accessing the filesystem as quickly as possible

Large Disk Usage

- On ARC systems you have two areas on the disk
- A small “home” area – this is where you log into
- A much larger “data” area
- Thus for your batch jobs we strongly suggest you use the data area
- The data area can be accessed from home via `cd $DATA`

```
Wot now? ssh -CX amg@arcus-b.arc.ox.ac.uk
amg@arcus-b.arc.ox.ac.uk's password:
Last login: Tue Jan 10 10:56:05 2018 from somewhere.at.ox.ac.uk
*****************************************************************
* Welcome to arcus-b - a Haswell IBM/Lenovo NeXtScale cluster    *
*                                                               *
* Please report all issues to support@arc.ox.ac.uk              *
*                                                               *
* /home/name (or $HOME) = Your (small) home area                *
* /data/group/name (or $DATA) = Your working area for data storage *
*****************************************************************
[amg@login12(arcus-b) ~]$ pwd
/home/amg
[amg@login12(arcus-b) ~]$ cd $DATA
[amg@login12(arcus-b) ijb]$ pwd
/data/myproject/amg
```

Efficient Disk Usage

- The filesystem on ARC systems is something called GPFS.
- This is a “high performance filesystem”
- *However* the way it works means that you will get best performance if you use a small number of large files accessing them in large chunks
  - Actually this is true for most filesystems
- Having and accessing a very large number of small files will cripple your performance on ARC
- We have had cases of users having 100,000+ small (a few kbyte) files all in one directory. This will lead to *very* slow performance
- And what is more as the disks are shared it’s not just slow for you, it’s slow for everybody!

A Final Word on Disks

- Please note the disks on ARC systems are NOT there for permanent storage
- They are not backed up
- After you have generated your data you should transfer it back to your “home machines” via sftp or similar mechanisms, and then delete it on ARC
- Also please note that as the filesystem is getting very full on ARC we are beginning to look at automatically deleting files that have not been used in a long time …

Measures of Performance

The most important measure

- In this section we will introduce a few measures of performance for computer codes, both serial and parallel
- But never forget that what you ultimately want to maximise is the amount of *science per second* that you generate
- This may be as simple as minimising the run time of your program
- But it may involve other factors
  - Using a more familiar application
  - Getting the best out of your computer budget
  - Turnaround on the cluster
    - **Higher core counts tend to turn around more slowly**

FLOPS

- A traditional measure of performance in computational science is Floating Point Operations per Second (FLOPS)
- Modern codes should be achieving GFLOPS+
- Can be useful but generally not because
  - Doesn't really measure what is crucial on todays computers
  - How many FLOPS is your method of solution capable of?
  - How many floating point operations did you computer actually do during your calculation?
    - Hardware counters can give an estimate

Parallel measures of performance

- Measuring the performance of parallel codes generally asks questions related to how much better are things running on multiple cores when compared to running on a single core or node
- We'll look at
  - Speed Up
  - Cost

Speed Up

- Speed up answers the question “How much faster does my program run if I use P cores”

$S(P)=T_1/T(P)$

- So if I use 100 processors it will run 100 times faster, right?
- And it can’t run more than 100 times faster, right?
- Also, I have 100 times as much memory so I can run 100 times bigger a problem, right?

Absolute Speed Up

- What we should really measure is Absolute Speed Up

$S(P)=T_s/T(P)$

- Where T_s is the time to run the best implementation of the serial program, and T(P) is the time to run the parallel code on P cores
  - You may use a different algorithm in the parallel code from the serial code

Relative Speed Up

- However what is almost always measured is Relative Speed Up

$S(P)=T(1)/T(P)$

- i.e. we compare the speed against the time taken on 1 core by the parallel program
- Saves writing both the serial and parallel code

Linear Speed Up

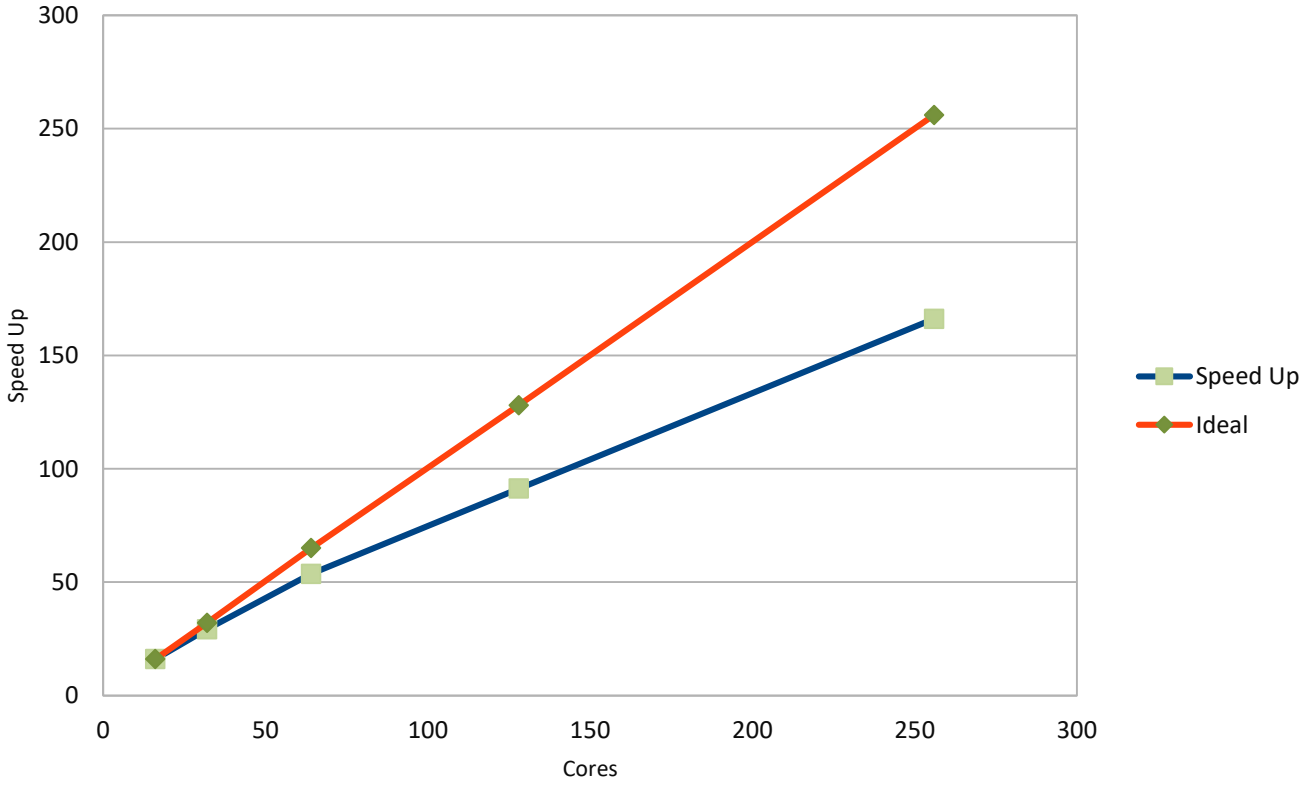
- Linear speed up is simply

$S(P)=P$

- So if you run on P processors, it runs P times quicker
- This is the ideal situation
- Also called perfect scaling

What Does it Look Like

- DL_POLY, 512,000 particles of NaCl on Arcus-B



So Why is it Not Perfect!?

- Many Possible Reasons!
- We've touched on load balance
- To go beyond this again is beyond what we are trying to cover here
- The main thing is NOT to expect perfect speed up
- And to be aware that using too many cores can actually DEGRADE your performance

Cost

- On some computers you will be CHARGED(!!) for use
- The usual situation on national resources
- Typically you get a finite budget which gets deducted from every time you use the machine
- Quite how the charging works will vary from machine to machine. Typically you get charged a fixed rate per node that you use, and for every second you use
  - Cost=nodes*time
  - This is the model on ARC machines
- i.e. If you leave cores idle in a node you get charged for them

So How Many Cores Should I Use

- Well, firstly don't put more processes or threads on a node than there are cores!!
- It depends …
- It depends on the code you are running
  - It may have special parallel options which you should learn about
- It depends on the case you are running
- It depends on the computer you are running on
- More cores will cost you more
- More cores may even slow you calculation down
- More cores will probably mean slower turn around
- It depends upon you and how important cost and turnaround are
- BUT DON'T JUST GUESS!
- One thing you can do is to run a little experiment

An Experiment

- Many Scientific codes do the same kind of thing many times
  - E.g. timesteps
  - E.g. iterations in a solver
- So plot the speed up curve for a few iterations and from that decide on a good number of cores for you
- For instance a full DL_POLY run will require at the very least many thousands of times steps
- So first run it for 100 timesteps on 1, 2, 4, 8, 16 … cores and see what the speed up curve looks like
- And then use that number of cores for the full run

Summary

- It's very difficult to predict the performance of a real application *a priori*
- So you will have to do experiments
- Many applications are iterative
- So measure the performance on a number of different cores for a small number of iterations and use that to work out what to use for the full run

Acknowledgments

We would like to thank the Numerical Algorithms Group Ltd. (NAG) for providing much of the background material used in the slides.

Reuse of this material